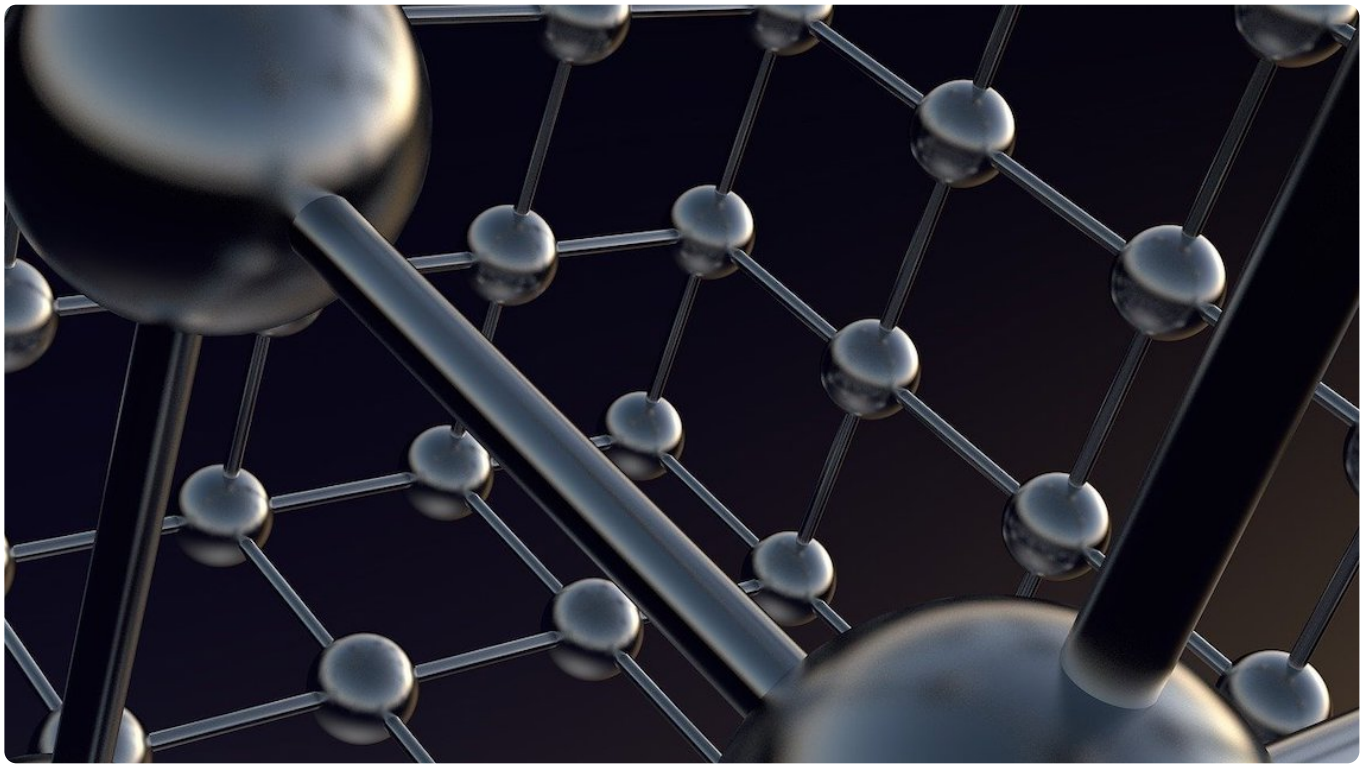
09｜原子性：2PC还是原子性协议的王者吗?

2020-08-28 王磊

分布式数据库30讲 进入课程 >



讲述：王磊
时长 25:27 大小 23.32M

你好，我是王磊，你也可以叫我 Ivan。今天，我要和你讲一讲分布式事务的原子性。

在限定“分布式”范围之前，我们先认识一下“事务的原子性”是啥。

如果分开来看的话，事务可以理解为包含一系列操作的序列，原子则代表不可分割的最小粒度。

而合起来看的话，事务的原子性就是让包含若干操作的事务表现得像一个最小粒度的操作。这个操作一旦被执行，只有“成功”或者“失败”这两种结果。这就好像比特（bit），只能代表 0 或者 1，没有其他选择。

为什么要让事务表现出原子性呢？我想举个从 ATM 取款的例子。

现在，你走到一台 ATM 前，要从自己 50,000 元的账户上取 1,000 元现金。当你输入密码和取款金额后， ATM 会吐出 1,000 块钱，同时你的账户余额会扣减 1,000 元；虽然有些时候，ATM 出现故障，无法吐钞，系统会提示取款失败，但你的余额还会保持在 50,000 元。

总之，要么既吐钞又扣减余额，要么既不吐钞又不扣减余额，你拿到手的现金和账户余额总计始终是 50,000 元，这就是一个具有原子性的事务。

显然，吐钞和扣减余额是两个不同的操作，而且是分别作用在 ATM 和银行的存款系统上。当事务整合了两个独立节点上的操作时，我们称之为分布式事务，其达成的原子性也就是分布式事务的原子性。

关于事务的原子性，图灵奖得主、事务处理大师詹姆斯·格雷（Jim Gray）给出了一个更权威的定义：

Atomicity. Either all the changes from the transaction occur (writes, and messages sent), or none occur.

这句话说得很精炼，我再和你解释下。

原子性就是要求事务只有两个状态：

一是成功，也就是所有操作全部成功；

二是失败，任何操作都没有被执行，即使过程中已经执行了部分操作，也要保证回滚这些操作。

要做到事务原子性并不容易，因为多数情况下事务是由多个操作构成的序列。而分布式事务原子性的外在表现与事务原子性一致，但前者要涉及多个物理节点，而且增加了网络这个不确定性因素，使得问题更加复杂。

实现事务原子性的两种协议

那么，如何协调内部的多项操作，对外表现出统一的成功或失败状态呢？这需要一系列的算法或协议来保证。

面向应用层的 TCC

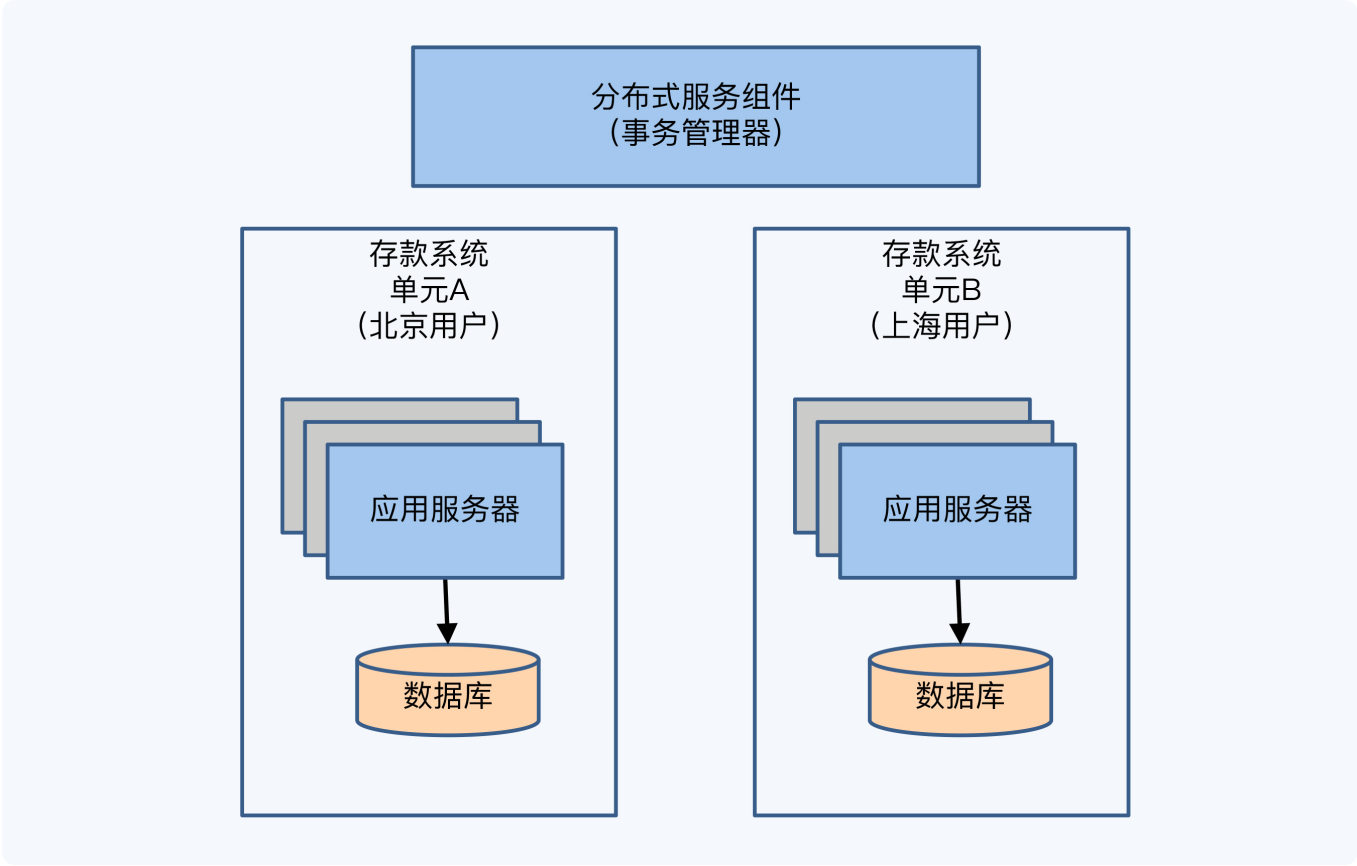
原子性提交协议有不少，按照其作用范围可以分为面向应用层和面向资源层。我想先给你介绍一种“面向应用层”中比较典型的协议，TCC 协议。

TCC 是 Try、Confirm 和 Cancel 三个单词的缩写，它们是事务过程中的三个操作。关于 TCC 的适用场景嘛，还记得我在第 1 讲中介绍的“单元架构 + 单体数据库”吗？这类方案需要在应用层实现事务的原子性，经常会用到 TCC 协议。

下面，我用一个转账的例子向你解释 TCC 处理流程。

小明和小红都是番茄银行的客户，现在小明打算给小红转账 2,000 元，这件事在番茄银行存款系统中是如何实现的呢？

我们先来看下系统的架构示意图：



显然，番茄银行的存款系统是单元化架构的。也就是说，系统由多个单元构成，每个单元包含了一个存款系统的部署实例和对应的数据库，专门为某一个地区的用户服务。比如，单元 A 为北京用户服务，单元 B 为上海用户服务。

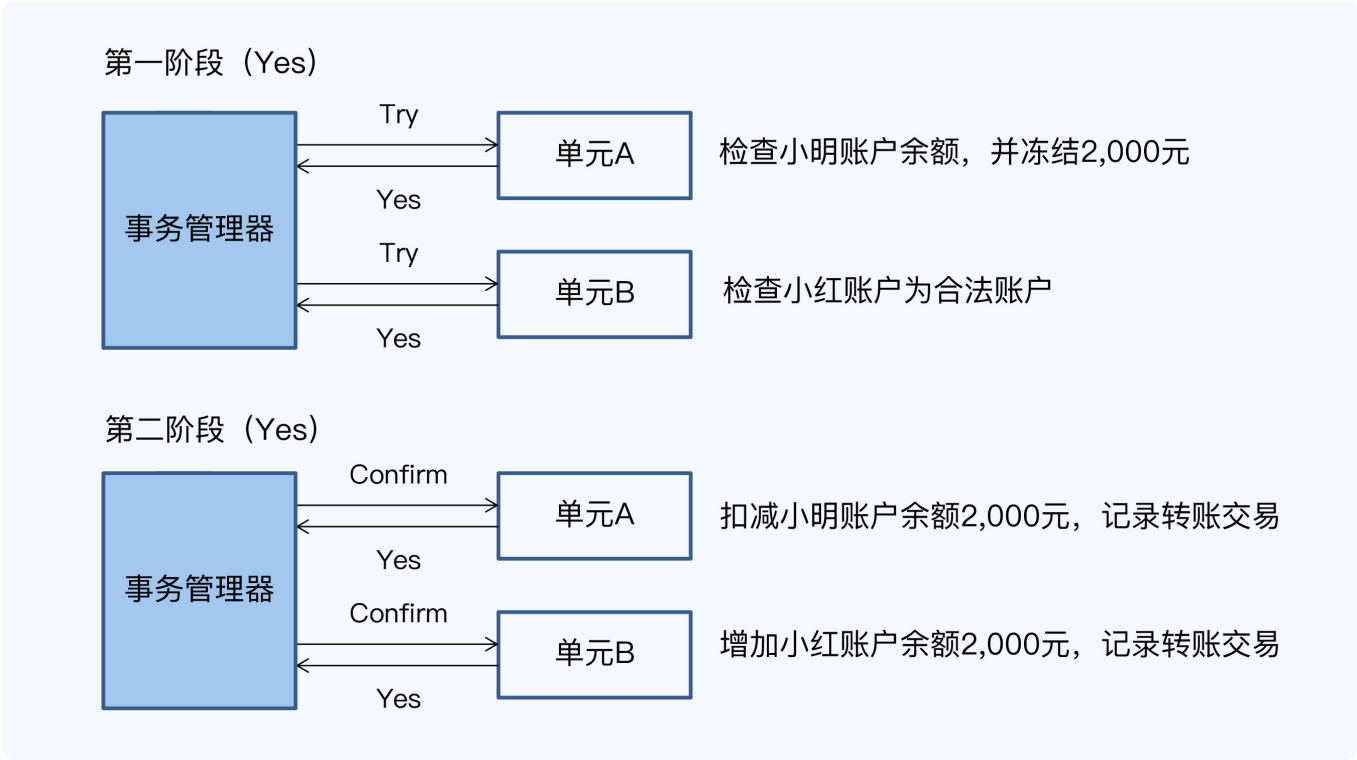
单元化架构的好处是每个单元只包含了部分用户，这样运行负载比较小，而且一旦出现问题，也只影响到少部分客户，可以提升整个存款系统的可靠性。

不过这种架构也有局限性。那就是虽然单元内的客户转账非常容易，但是跨单元的转账需要引入额外的处理机制，而 TCC 就是一种常见的选择。

TCC 的整个过程由两类角色参与，一类是事务管理器，只能有一个；另一类是事务参与者，也就是具体的业务服务，可以是多个，每个服务都要提供 Try、Confirm 和 Cancel 三个操作。

下面是 TCC 的具体执行过程。

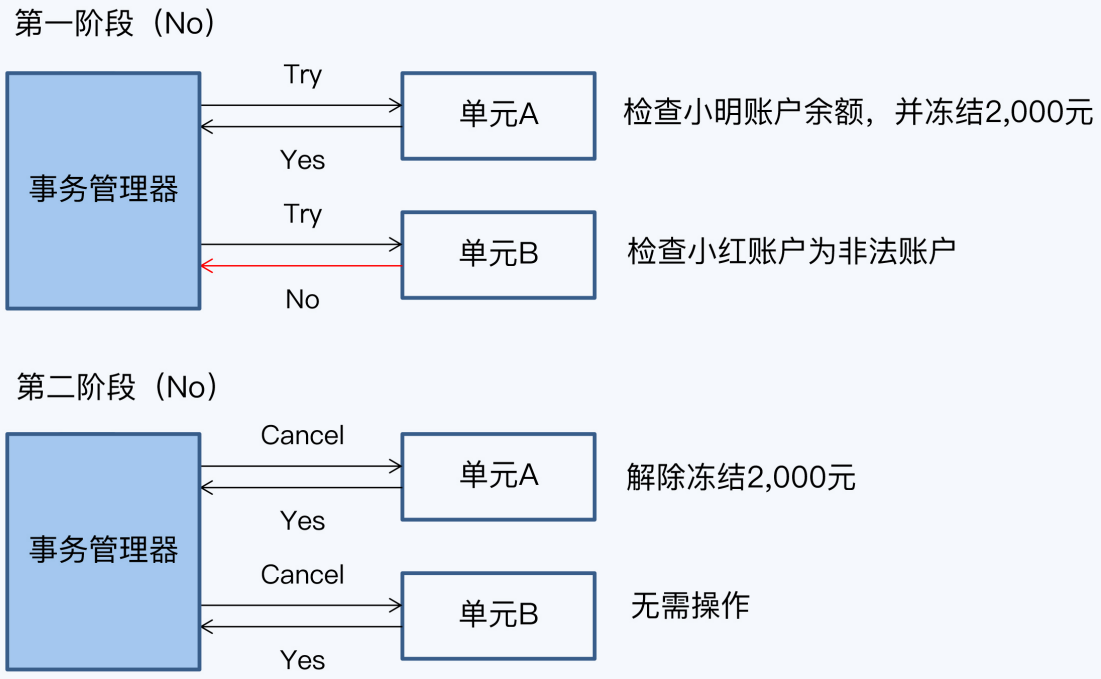
小明的银行卡在北京的网点开户，而小红的银行卡是在上海出差时办理的，所以两人的账户分别在单元 A 和单元 B 上。现在小明的账户余额是 4,900 元，要给小红转账 2,000 元，一个正常流程是这样的。



第一阶段，事务管理器会发出 Try 操作，要求进行资源的检查和预留。也就是说，单元 A 要检查小明账户余额并冻结其中的 2,000 元，而单元 B 要确保小红的账户合法，可以接收转账。在这个阶段，两者账户余额始终不会发生变化。

第二阶段，因为参与者都已经做好准备，所以事务管理器会发出 Confirm 操作，执行真正的业务，完成 2,000 元的划转。

但是很不幸，小红账户是无法接收转账的非法账户，处理过程就变成下面的样子。



第一阶段，事务管理器发出 Try 指令，单元 B 对小红账户的检查没有通过，回复 No。而单元 A 检查小明账户余额正常，并冻结了 2,000 元，回复 Yes。

第二阶段，因为前面有参与者回复 No，所以事务管理器向所有参与者发出 Cancel 指令，让已经成功执行 Try 操作的单元 A 执行 Cancel 操作，撤销在 Try 阶段的操作，也就是单元 A 解除 2,000 元的资金冻结。

从上述流程可以发现，**TCC 仅是应用层的分布式事务框架**，具体操作完全依赖于业务编码实现，可以做针对性的设计，但是这也意味着业务侵入会比较深。

此外，考虑到网络的不可靠，操作指令必须能够被重复执行，这就要求 Try、Confirm、Cancel 必须是幂等性操作，也就是说，要确保执行多次与执行一次得到相同的结果。显然，这又增加了开发难度。

那还有其他的选择吗?

当然有，我们来看看数据库领域最常用的两阶段提交协议（Two-Phase Commit，2PC），这也是面向资源层的典型协议。

数据库领域最常用的 2PC

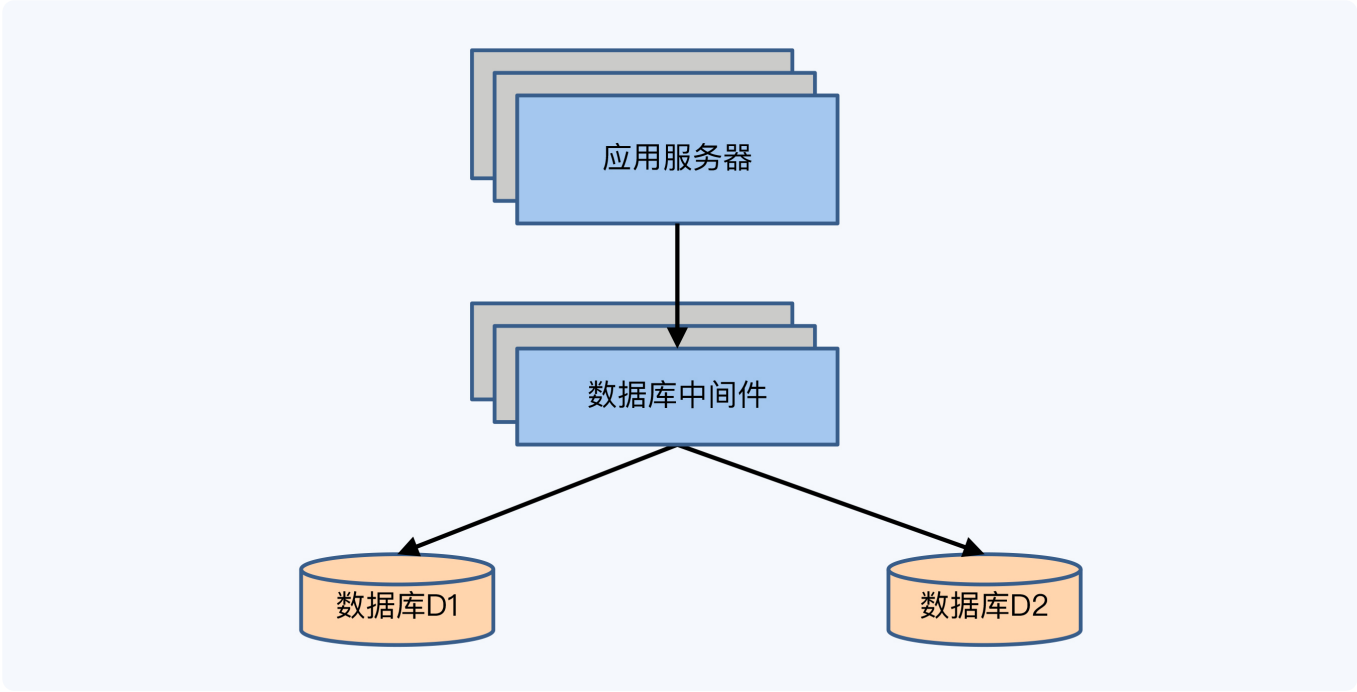
2PC 的首次正式提出是在 Jim Gray 1977 年发表的一份文稿中，文稿的题目是“Notes on Data Base Operating Systems”，对当时数据库系统研究成果和实践进行了总结，而 2PC 在工程中的应用还要再早上几年。

2PC 的处理过程也分为准备和提交两个阶段，每个阶段都由事务管理器与资源管理器共同完成。其中，事务管理器作为事务的协调者只有一个，而资源管理器作为参与者执行具体操作允许有多个。

2PC 具体是如何运行的呢？我们还是说回小明转账的例子。

小明给小红转账没有成功，两人又到木瓜银行来尝试。

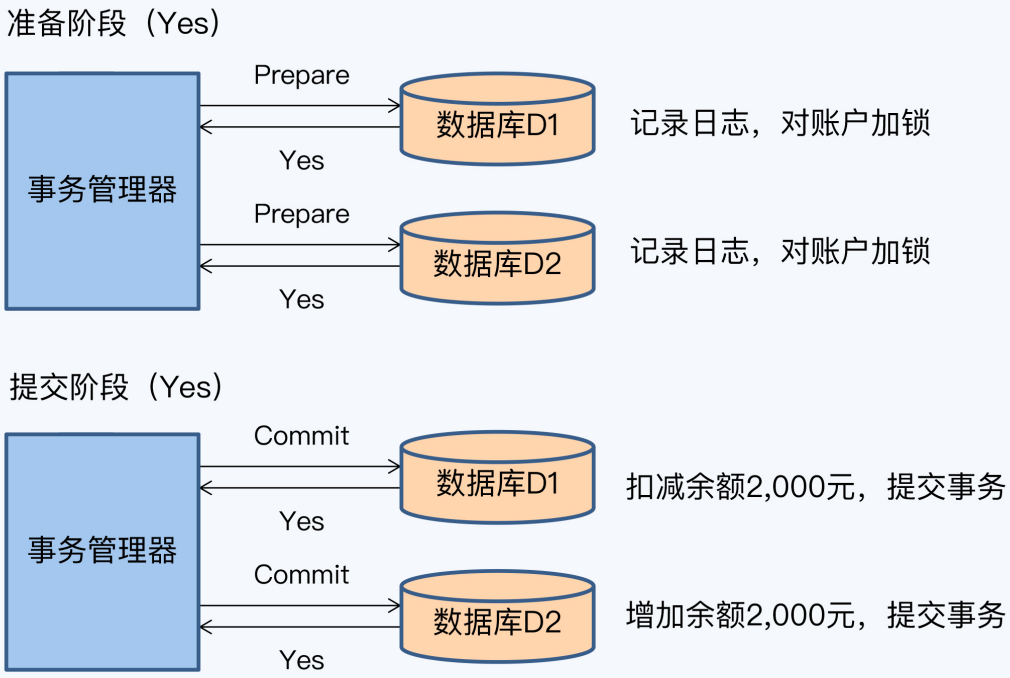
木瓜银行的存款系统采用了分库分表方案，系统架构大致是这样的：



在木瓜银行的存款系统中，所有客户的数据被分散存储在多个数据库实例中，这些数据库实例具有完全相同的表结构。业务逻辑部署在应用服务器上，通过数据库中间件访问底层的数据库实例。数据库中间件作为事务管理器，资源管理器就是指底层的数据库实例。

假设，小明和小红的数据分别被保存在数据库 D1 和 D2 上。

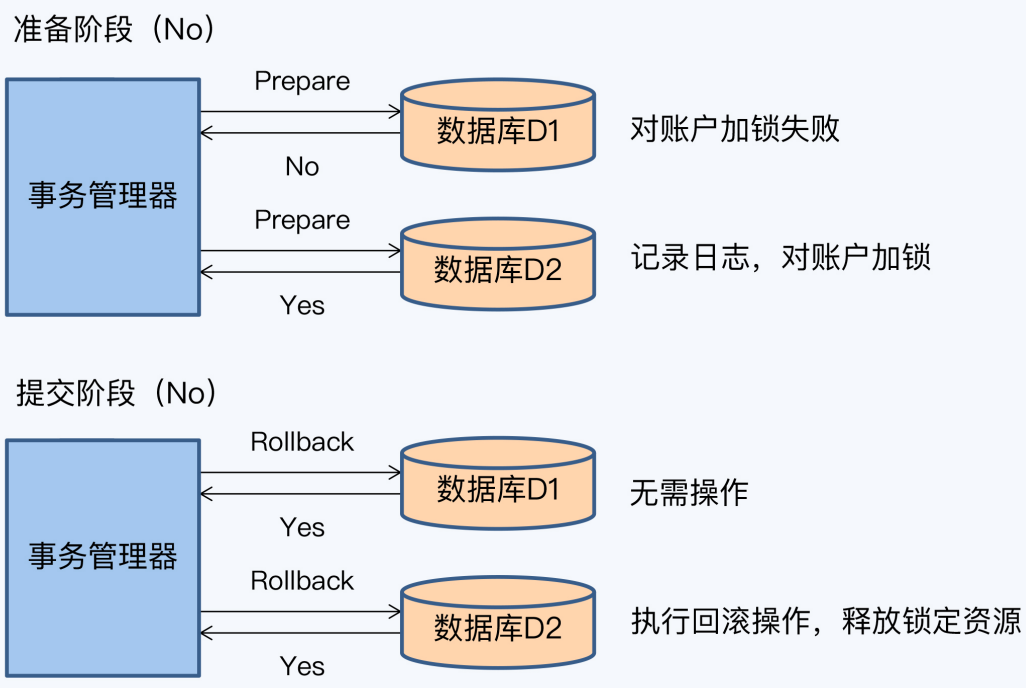
我们还是先讲正常的处理流程。



第一阶段是准备阶段，事务管理器首先向所有参与者发送待执行的 SQL，并询问是否做好提交事务的准备（Prepare）；参与者记录日志、分别锁定了小明和小红的账户，并做出应答，协调者接收到反馈 Yes，准备阶段结束。

第二阶段是提交阶段，如果所有数据库的反馈都是 Yes，则事务管理器会发出提交（Commit）指令。这些数据库接受指令后，会进行本地操作，正式提交更新余额，给小明的账户扣减 2,000 元，给小红的账户增加 2,000 元，然后向协调者返回 Yes，事务结束。

那如果小明的账户出了问题，导致转账失败，处理过程会是怎样呢？



第一阶段，事务管理器向所有数据库发送待执行的 SQL，并询问是否做好提交事务的准备。

由于小明之前在木瓜银行购买了基金定投产品，按照约定，每月银行会自动扣款购买基金，刚好这个自动扣款操作正在执行，先一步锁定了账户。数据库 D1 发现无法锁定小明的账户，只能向事务管理器返回失败。

第二阶段，因为事务管理器发现数据库 D1 不具备执行事务的条件，只能向所有数据库发出“回滚”（Rollback）指令。所有数据库接收到指令后撤销第一阶段的操作，释放资源，并向协调者返回 Yes，事务结束。小明和小红的账户余额均保持不变。

2PC 的三大问题

学完了 TCC 和 2PC 的流程，我们来对比下这两个协议。

相比于 TCC，2PC 的优点是借助了数据库的提交和回滚操作，不侵入业务逻辑。但是，它也存在一些明显的问题：

1. **同步阻塞**

执行过程中，数据库要锁定对应的数据行。如果其他事务刚好也要操作这些数据行，那它们就只能等待。其实同步阻塞只是设计方式，真正的问题在于这种设计会导致分布式事务出现高延迟和性能的显著下降。

2. **单点故障**

事务管理器非常重要，一旦发生故障，数据库会一直阻塞下去。尤其是在第二阶段发生故障的话，所有数据库还都处于锁定事务资源的状态中，从而无法继续完成事务操作。

3. **数据不一致**

在第二阶段，当事务管理器向参与者发送 Commit 请求之后，发生了局部网络异常，导致只有部分数据库接收到请求，但是其他数据库未接到请求所以无法提交事务，整个系统就会出现数据不一致性的现象。比如，小明的余额已经能够扣减，但是小红的余额没有增加，这样就不符合原子性的要求了。

你可能会问：**这些问题非常致命呀，2PC 到底还能不能用？**

所以，网上很多文章会建议你避免使用 2PC，替换为 TCC 或者其他事务框架。

但我要告诉你的是，别轻易放弃，2PC 都提出 40 多年了，学者和工程师们也没闲着，已经有很多对 2PC 的改进都在不同程度上解决了上述问题。

事实上，多数分布式数据库都是在 2PC 协议基础上改进，来保证分布式事务的原子性。这里我挑选了两个有代表性的 2PC 改进模型和你展开介绍，它们分别来自分布式数据库的两大阵营，NewSQL 和 PGXC。

分布式数据库的两个 2PC 改进模型

NewSQL 阵营：Percolator

首先，我们要学习的是 NewSQL 阵营的 Percolator。

Percolator 来自 Google 的论文“Large-scale Incremental Processing Using Distributed Transactions and Notifications”，因为它是基于分布式存储系统 BigTable

建立的模型，所以可以和 NewSQL 无缝链接。

Percolator 模型同时涉及了隔离性和原子性的处理。今天，我们主要关注原子性的部分，在讲并发控制时，我再展开隔离性的部分。

使用 Percolator 模型的前提是事务的参与者，即数据库，要**支持多版本并发控制（MVCC）**。不过你不用担心，现在主流的单体数据库和分布式数据库都是支持的 MVCC。

在转账事务开始前，小明和小红的账户分别存储在分片 P1 和 P2 上。如果你不了解分片的含义，可以回到第 6 讲学习。当然，你也可以先用单体数据库来替换分片的概念，这并不会妨碍对流程的理解。

分片 P1

Key	Bal. data	Bal. lock	Bal. write
Ming	6:		data @5
Ming	5: 4900		

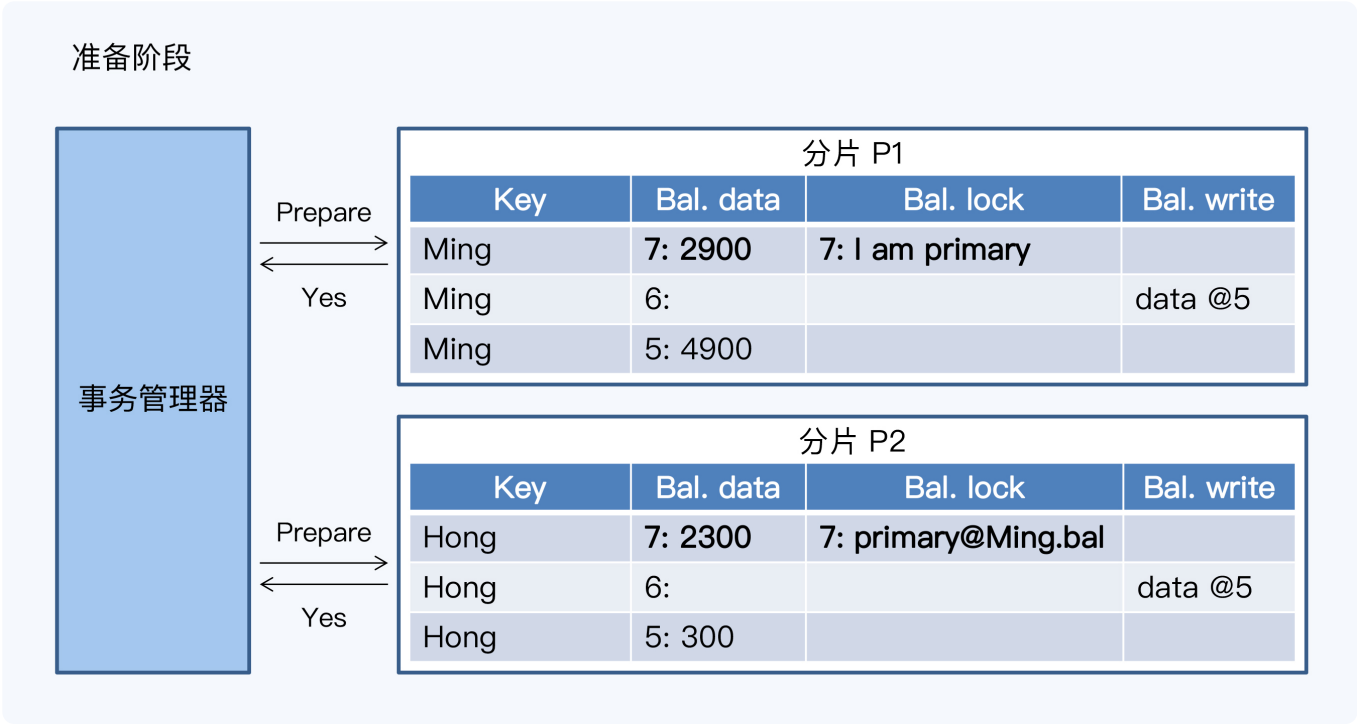
分片 P2

Key	Bal. data	Bal. lock	Bal. write
Hong	6:		data @5
Hong	5: 300		

上图中的 Ming 代表小明，Hong 代表小红。在分片的账户表中各有两条记录，第一行记录的指针（write）指向第二行记录，实际的账户余额存储在第二行记录的 Bal. data 字段中。

Bal.data 分为两个部分，冒号前面的是时间戳，代表记录的先后次序；后面的是真正的账户余额。我们可以看到，现在小明的账户上有 4,900 元，小红的账户上有 300 元。

我们来看下 Percolator 的流程。



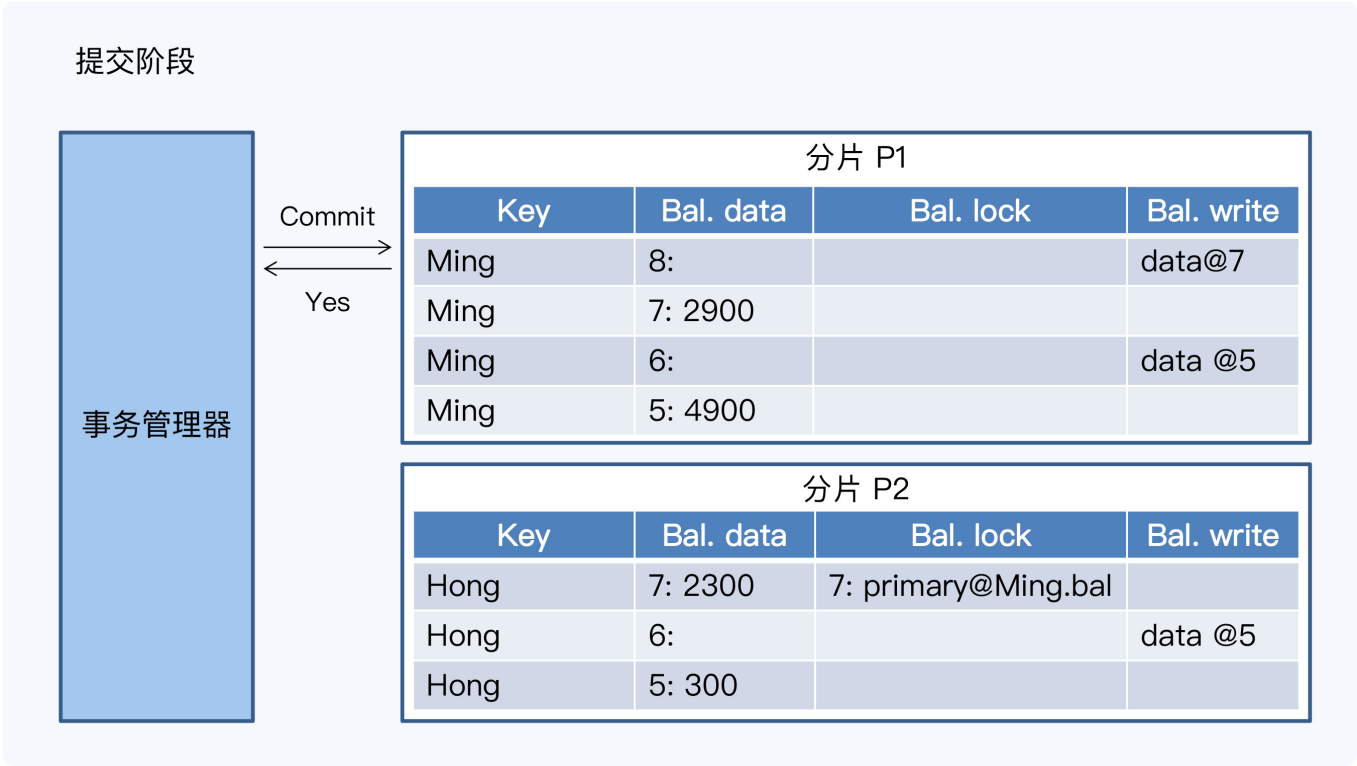
第一，准备阶段，事务管理器向分片发送 Prepare 请求，包含了具体的数据操作要求。

分片接到请求后要做两件事，写日志和添加私有版本。关于私有版本，你可以简单理解为，在 lock 字段上写入了标识信息的记录就是私有版本，只有当前事务能够操作，通常其他事务不能读写这条记录。

你可能注意到了，两个分片上的 lock 内容并不一样。

主锁的选择是随机的，参与事务的记录都可能拥有主锁，但一个事务只能有一条记录拥有主锁，其他参与事务的记录在 lock 字段记录了指针信息“primary@Ming.bal”，指向主锁记录。

准备阶段结束的时候，两个分片都增加了私有版本记录，余额正好是转账顺利执行后的数字。



第二，提交阶段，事务管理器只需要和拥有主锁的分片通讯，发送 Commit 指令，且不用附带其他信息。

分片 P1 增加了一条新记录时间戳为 8，指向时间戳为 7 的记录，后者在准备阶段写入的主锁也被抹去。这时候 7、8 两条记录不再是私有版本，所有事务都可以看到小明的余额变为 2,700 元，事务结束。

你或许要问，为什么在提交阶段不用更新小红的记录?

Percolator 最有趣的设计就是这里，因为分片 P2 的最后一条记录，保存了指向主锁的指针。其他事务读取到 Hong7 这条记录时，会根据指针去查找 Ming.bal，发现记录已经提交，所以小红的记录虽然是私有版本格式，但仍然可视为已经生效了。

当然，这种通过指针查找的方式，会给读操作增加额外的工作。如果每个事务都照做，性能损耗就太大了。所以，还会有其他异步线程来更新小红的余额记录，最终变成下面的样子。

分片 P1

Key	Bal. data	Bal. lock	Bal. write
Ming	8:		data@7
Ming	7: 2900		
Ming	6:		data @5
Ming	5: 4900		

分片 P2

Key	Bal. data	Bal. lock	Bal. write
Hong	8:		data@7
Hong	7: 2300		
Hong	6:		data @5
Hong	5: 300		

现在，让我们对比 2PC 的问题，来看看 Percolator 模型有哪些改进。

1. **数据不一致**

2PC 的一致性问题主要缘自第二阶段，不能确保事务管理器与多个参与者的通讯始终正常。

但在 Percolator 的第二阶段，事务管理器只需要与一个分片通讯，这个 Commit 操作本身就是原子的。所以，事务的状态自然也是原子的，一致性问题被完美解决了。

2. **单点故障**

Percolator 通过日志和异步线程的方式弱化了这个问题。

一是，Percolator 引入的异步线程可以在事务管理器宕机后，回滚各个分片上的事务，提供了善后手段，不会让分片上被占用的资源无法释放。

二是，事务管理器可以用记录日志的方式使自身无状态化，日志通过共识算法同时保存在系统的多个节点上。这样，事务管理器宕机后，可以在其他节点启动新的事务管理器，基于日志恢复事务操作。

Percolator 模型在分布式数据库的工程实践中被广泛借鉴。比如，分布式数据库 TiDB，完全按照该模型实现了事务处理；CockroachDB 也从 Percolator 模型获得灵感，设计了自己的 2PC 协议。

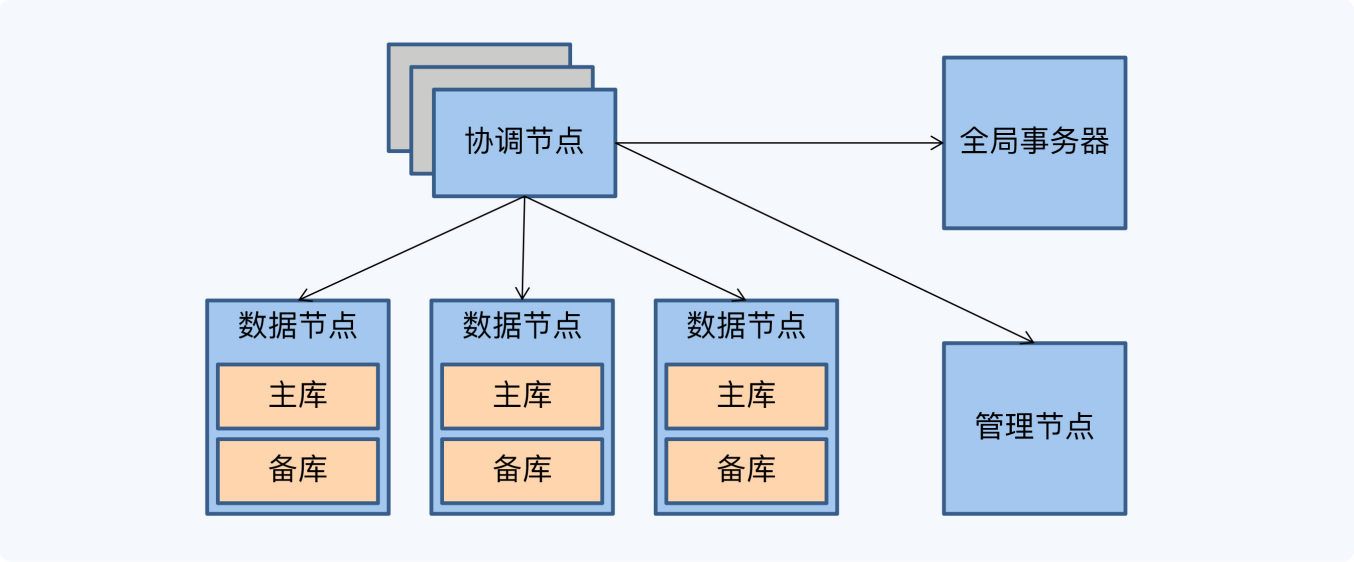
CockroachDB 的变化在于没有随机选择主锁，而是引入了一张全局事务表，所有分片记录的指针指向了这个事务表中对应的事务记录。单就原子性处理来说，这种设计似乎差异不大，但在相关设计上会更有优势，具体是什么优势呢，下一讲我来揭晓答案。

PGXC 阵营：GoldenDB 的一阶段提交

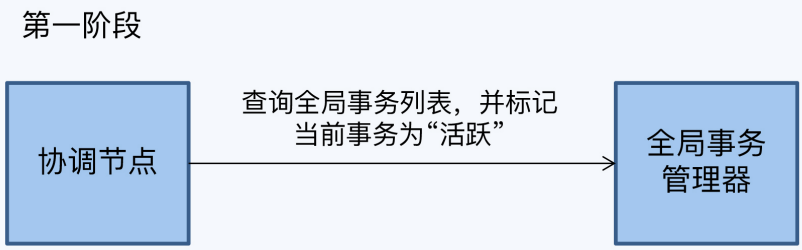
那么，分布式数据库的另一大阵营，PGXC，又如何解决 2PC 的问题呢?

GoldenDB 展现了另外一种改良思路，称之为“一阶段提交”。

GoldenDB 遵循 PGXC 架构，包含了四种角色：协调节点、数据节点、全局事务器和管理节点，其中协调节点和数据节点均有多个。GoldenDB 的数据节点由 MySQL 担任，后者是独立的单体数据库。

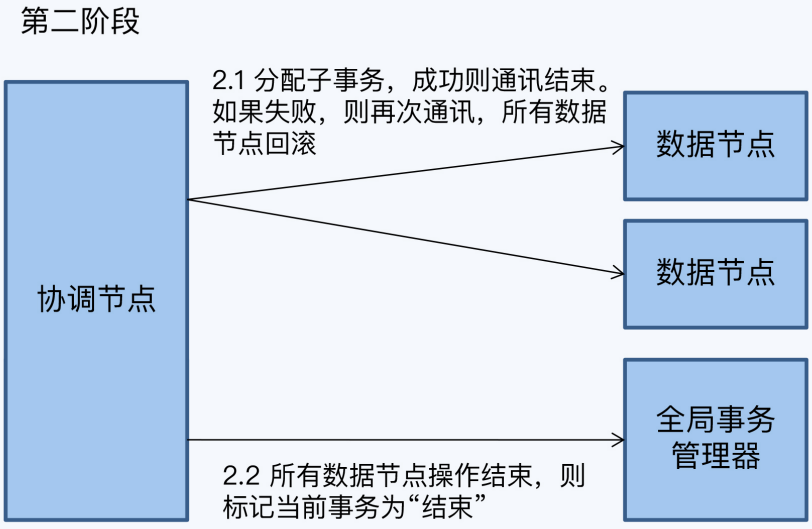


虽然名字叫“一阶段提交”，但 GoldenDB 的流程依然可以分为两个阶段。



第一阶段，GoldenDB 的协调节点接到事务后，在全局事务管理器（GTM）的全局事务列表中将事务标记成活跃的状态。这个标记过程是 GoldenDB 的主要改进点，实质是通过全局事务列表来申请资源，规避可能存在的事务竞争。

这样的好处是避免了与所有参与者的通讯，也减少了很多无效的资源锁定动作。



第二阶段，协调节点把一个全局事务分拆成若干子事务，分配给对应的 MySQL 去执行。如果所有操作成功，协调者节点会将全局事务列表中的事务标记为结束，整个事务处理完成。如果失败，子事务在单机上自动回滚，而后反馈给协调者节点，后者向所有数据节点下发回滚指令。

由于 GoldenDB 属于商业软件，公开披露信息有限，我们也就不再深入细节了，你只要能够理解上面我讲的两个阶段就够了。

GoldenDB 的“一阶段提交”，本质上是改变了资源的申请方式，更准确的说法是，并发控制手段从锁调度变为时间戳排序（Timestamp Ordering）。这样，在正常情况下协调

节点与数据节点只通讯一次，降低了网络不确定性的影响，数据库的整体性能有明显提升。因为第一阶段不涉及数据节点的操作，也就弱化了数据一致性和单点故障的问题。

小结

好了，以上就是今天的主要内容了，我希望你能记住以下几点：

1. 事务的原子性就是让包含若干操作的事务表现得像一个最小粒度的操作，而这个操作一旦被执行只有两种结果，成功或者失败。
2. 相比于单机事务，分布式事务原子性的复杂之处在于增加了多物理设备和网络的不确定性，需要通过一定的算法和协议来实现。这类协议也有不少，我重点介绍了 TCC 和 2PC 这两个常用协议。
3. TCC 提供了一个应用层的分布式事务框架，它对参与者没有特定要求，但有较强的业务侵入；2PC 是专为数据库这样的资源层设计的，不侵入业务，也是今天分布式数据库主流产品的选择。
4. 考虑到 2PC 的重要性和人们对其实用价值的误解，我又展开说明 2PC 的两种改良模型，分别是 Percolator 和 GoldenDB 的“一阶段提交”。Percolator 将 2PC 第二阶段工作简化到极致，减少了与参与者的通讯，完美解决了一致性问题，同时通过日志和异步线程弱化了单点故障问题。GoldenDB 则改良了 2PC 第一阶段的资源协调过程，将协调者与多个参与者的交互转换为协调者与全局事务管理器的交互，同样达到了减少通讯的效果，弱化了一致性和单点故障的问题。

这节课马上就要结束了，你可能要问，为什么咱们没学三阶段提交协议（Three-Phase Commit，3PC）呢?

原因也很简单，因为 3PC 虽然试图解决 2PC 的问题，但它的通讯开销更大，在网络分区时也无法很好地工作，很少在工程实践中使用，所以我就没有介绍，你只要知道有这么个协议就好。

另外，我还要提示一个容易与 2PC 协议混淆的概念，也就是两阶段封锁协议（Two-Phase Locking，2PL）。

我认为，这种混淆并不只是因为名字相似。从整个分布式事务看，原子性协议之外还有一层隔离性协议，由后者保证事务能够成功申请到资源。在相当长的一段时间里，2PC 与

2PL 的搭配都是一种主流实现方式，可能让人误以为它们是可以替换的术语。实际上，两者是截然不同的，2PC 是原子性协议，而 2PL 是一种事务的隔离性协议，也是一种并发控制算法。

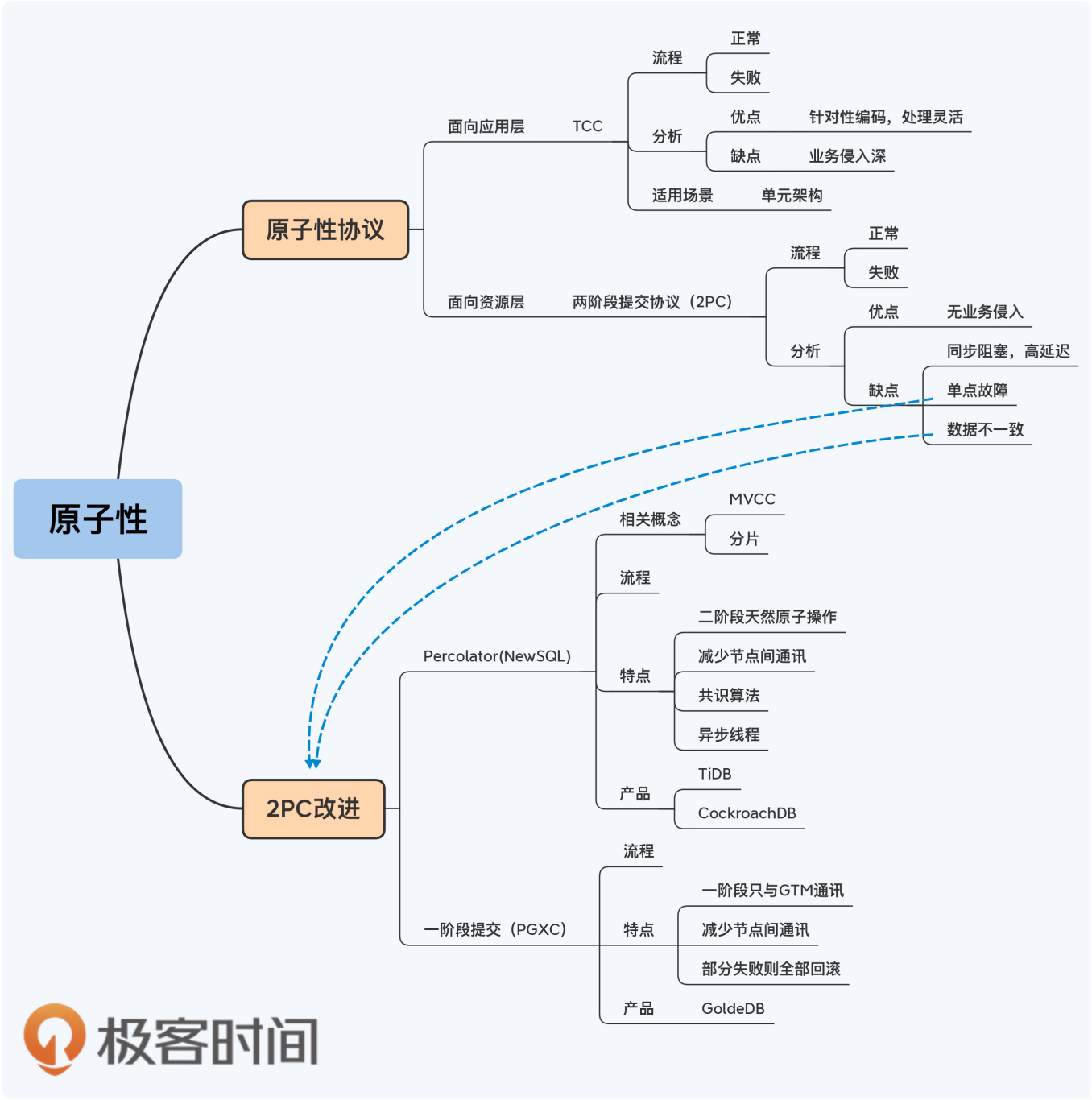
在这一节中，其实我们多次提到了并发控制算法，但都没有展开介绍，原因是这部分内容确实比较复杂，没办法用三言两语说清，我会在后面第 13 讲和第 14 讲中详细解释。

两种改良模型都一定程度上化解了 2PC 的单点故障和数据一致性问题，但同步阻塞导致的性能问题还没有根本改善，而这也是 2PC 最被诟病的地方，可能也是很多人放弃分布数据库的理由。

可是，2PC 注定就是延时较长、性能差吗？或者说分布式数据库中的分布式事务，延时一定很长吗？

我想告诉你的是，其实不少优秀的分布式数据库产品已经大幅缩短了 2PC 的延时，无论是理论模型还是工程实践都已经过验证。

那么，它们又有哪些精巧构思呢？我将在下一讲为你介绍这些黑科技。



思考题

最后，我给你留下一个思考题。今天内容主要围绕着 2PC 展开，而它的第一阶段“准备阶段”也被称为“投票阶段”，“投票”这个词是不是让你想到 Paxos 协议呢？

那么，你觉得 2PC 和 Paxos 协议有没有关系，如果有又是什么关系呢？

如果你想到了答案，又或者是触发了你对相关问题的思考，都可以在评论区和我聊聊，我会在下一讲和你一起探讨。最后，谢谢你的收听，希望这节课能带给你一些收获，欢迎你把它分享给周围的朋友，一起进步。

学习资料

Daniel Peng and Frank Dabek: *Large-scale Incremental Processing Using Distributed Transactions and Notifications*

Jim Gray: *Notes on Data Base Operating Systems*

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 基础篇大串讲：重难点回顾+思考题答疑+知识全景图

下一篇 10 | 原子性：如何打破事务高延迟的魔咒?

精选留言 (16)

写留言

Lost Horizon
2020-09-03

请问老师，TCC 第二阶段
1. 如果向单元 A 发出 confirm 操作成功且收到成功应答，但向单元 B 发出 confirm 操作失败，这时是否需要通过其它手段来回滚（或者补偿） A 的变更呢?
2. 如果向单元 A 发出 confirm 操作成功且收到成功应答，向单元 B 发出 confirm 操作成功，但没有收到成功应答，是否应该先确认 B 的状态，然后再决定是否需要回滚（或者...

展开

作者回复: 问题1，如果单元A成功，单元B失败，则面临一个选择。如果以努力成功为目标，则可以再次向单元B发送指令，有可能成功，这也是为什么要求操作具有幂等性。当然也可以直接撤销A操作，是A与B一样都是失败状态，这种就是失败回滚。
问题2，没有收到应答，其实也和问题1类似，重试争取成功，或者直接回滚。但这里面有个问题是，如果无论发送任何指令，单元B始终也没有返回，怎么办。这个时候基本上就是要人工干预了。当然，对于这种网络通讯故障也不是没有解决办法，有种协议叫做Paxos Commmit，可以极大缓解这个问题，我会在第15讲答疑篇详细介绍，你可以关注下。

2 3

扩散性百万咸面包
2020-08-28

老师不是很理解为什么TCC就不用像2PC那样加锁和记日志了呢？TCC如何保证事务隔离性呢？如果有其他代码修改同一行数据怎么办？

展开

作者回复: TCC可使用的手段更灵活，不限于数据库锁了。比如，可以增加一个“未冻结余额”字段，初始值和余额一样，一阶段时直接在这个字段上扣减金额，这样后发生的事务如果发现剩下的“未冻结余额”不够，就会返回失败，这样多事务就可以协同了。

2 2

piboye
2020-08-28

paxos是共识算法，是对同一份数据的达成共识。2pc更多是为了达成的多份不同数据修改的原子性。不知道这样理解对不?

展开

作者回复: 是的，赞。还有延伸内容，同样建议关注第15讲。

1

chenchukun
2020-08-28

个人的理解是paxos可以用于解决2PC的单点故障和数据不一致问题，协调者和参与者利用paxos实现多副本一致，在节点宕机后可切换到副本节点继续完成2PC流程。
今天讲的内容，Percolator很好理解，很好的解决了传统2PC存在的问题。
但是对于PGXC的解决方案还是不明白，几个问题请教一下老师，或者请老师帮忙推荐一下相关学习资料。...
展开

1

扩散性百万咸面包
2020-09-10

感觉老师这里可以对Percolator的设计思想再阐述得更详细一些：
1. 为什么需要MVCC来实现percolator?或者说为什么要保存多个版本的key才能实现percolator?
2. 为什么要有个write指向实际数据存储的行？而不是直接存储对应数据？
展开

作者回复: 第一个问题，如果只实现“用主锁来控制事务的原子性，从锁保留指向主锁的指针”这个设计，那确实不一定要用MVCC。Percolator是一个工业实现，它要解决自己的业务需求，除了原子性还有隔离性要处理，使用了MVCC还可以解决读写冲突嘛。
第二个问题，我觉得是设计细节，采用其他形式应该也是可以的。

扩散性百万咸面包
2020-09-07

这个思考题我延伸到Raft思考一下：
Raft里面基于Leader的复制是否就是一种2PC呢？有写请求时，Leader先发给大多数节点，成功再写入。但是又好像不完全一样。面试的时候面试官也说这不是2PC。
展开

piboye
2020-09-06

paxos提供可靠性，2pc提供原子性，所以3pc没优势？

展开

piboye
2020-09-06

tcc和goden的方式隔离性有问题吧？ 都可能出现读中间状态的情况

作者回复: 理论上是不会的。GoldenDB是通过全局事务列表统一控制，数据节点（也就是单体数据库）基于日志实现回滚。TCC只要三个操作都正确实现，理论上也不会出现中间状态。

Bryant.C
2020-09-04

自己认为2pc更像是一种思想，多个节点对某件事达成共识，其实paxos完成的也是类似的行为，就单个操作在多副本上达成一致或共识，因此借鉴了2pc思想

展开

myrfy
2020-09-03

老师，GlodenDB如何避免全局管理器成为瓶颈呢？

展开

作者回复: 从公开资料看，也没有太特别的设计，就是把这个进程独立部署，这样不会与其他任务竞争资源。

南国
2020-09-01

感觉2PC和Basic-Paxos的过程好像啊，第一阶段的区别是2PC需要全部的回复，而Paxos只需要一半以上的Acceptor回复；第二阶段就几乎一模一样了。至于为什么第一阶段有这样的区别，大概是2PC的每一个节点职能（包括数据）都不相同，要满足一致性约束必须全部的节点的同意；而Paxos抛去每个节点角色不同，它们存储的数据都一样（理想中一致，实际不一致，Paxos会出现日志空缺），为了全局一致，一次同意一半以上就可以...

展开

作者回复: 我觉得2PC还不是共识算法，因为参与者并不是对一件事情（一个状态）投票，而是要让多个独立的状态要保持一致，且参与者并不了解这种一致性。可以参照15讲的Paxos Commit协议再体会下。

qinsi
2020-08-30

关于Percolator，文中提到“在 lock 字段上写入了标识信息的记录就是私有版本，只有当前事务能够操作”，而在例子中又有其他事务读到了私有版本的数据，这是为什么呢?

作者回复: 如果加了secondary锁，其他事务确实会读取到，是为了追溯事务状态，再决定读取哪个版本。而在异步线程或第一个读操作维护好状态后，就不再是私有版本了，所以我们说“通常其他事务不能读写”

tt
2020-08-29

还有，最近有一个业务，涉及到两个的系统，业务也要求两个系统必须都成功或者都失败，通过报文交互，正好适合TCC这种业务层面的协议

展开

作者回复: 嗯，这个场景比较适合TCC

tt
2020-08-29

我的理解是2PC中的任何一个节点都是逻辑的，可以有多个物理节点组成，PAXOS可以让这个逻辑节点内部的值达成一致，就是各种一致性模型。

而2PC是解决分布式的事物，参与的各个节点的值是不一样的，比如小明节点的余额是2900，而小红节点的余额是2300

展开

李鑫磊

2020-08-28

2PC 解决的是：
1）小明账户 - 100；
2）小红账户 + 100；
3）小明和小红账户信息存储在不同的数据库实例中；
Basic Paxos 解决的问题：...

展开

作者回复: 是的，点赞。不过两者也可以联系起来看，我在第15讲会有详细说明。

伟伟哦
2020-08-28

老师课程 TCC、3cp、2pc这些，或者以后课程， 可以接入下代码吗?

1